V-Oat Design Principles

Seungmin Han

April 16, 2023

1 Abstract

This document is a more technical overview of V-Oat's system and algorithm design. It is intended to clarify as much as possible what is actually going on behind the scenes of a seemingly-simple vote counter platform to achieve zero information leaks during the entire voting process. Three major design principles are emphasized: the two-client system, the user verification procedure, and the vote verification algorithm. The document ends with outlining the realistic limitations of V-Oat design and provides directions for possible future work.

2 Introduction

2.1 Overview of V-Oat

V-Oat aims to provide a novel approach to a secure online voting platform using homomorphic encryption. Online voting platforms have long been desired due to their efficiency and convenience, but factors such as privacy and security have been major deterring factors against them. V-Oat prototypes a fully private and secure voting system using homomorphic encryption technology that allows arithmetic operations between encrypted ciphertexts. User information, votes, and ballots are all encrypted throughout the entire voting process until the ballot is closed and decrypted to display the results. We designed specific algorithms to ensure user and vote verification while revealing no information about the user or the vote throughout the process. With V-Oat, we believe we are one step closer to a future of a fully online voting procedure. `V-Oat` is open source and uploaded on Github.

2.2 Overview of V-Oat Design

The Two-Client System Design limits the V-Oat server's abilities as a computation-only server. This means that V-Oat server can count up the encrypted votes into the encrypted ballots, but cannot decrypt the information stored inside these votes and ballots. Instead, we relay this responsibility to the initial client, which we call the election maintainer. The election maintainer, however, will only be able to decrypt selective information given by the V-Oat server, which will not reveal any information about the user or the vote.

The User Verification Procedure defines a protocol that checks if a new client connection is

1. A registered client to vote for the ballot
2. Provided valid credentials
3. Has not voted yet

This procedure follows the principles of Private Information Retrieval and reveals no information about the user to the server throughout the procedure.

The Vote Verification Procedure defines a procedure that checks if a vote sent from the client was

1. Only modified once by the client
2. Follows the correct vote format, thus is a valid vote safe and ready to be cast

We aim to perform checks on votes homomorphically, so that no information about the vote is gained by the server except for the fact that the vote is a valid vote and is safe to be casted to the ballot. We leverage the abilities of homomorphic encryption scheme and functions that our selected library provides us to achieve such goal.

3 Background and Notation

We are using the Brakerski Gentry-Vaikuntanathan (BGV) Homomorphic Encryption (HE) scheme implemented in the `HElib` library [1, 2]. In this section, we introduce notations used throughout the document.

3.1 Plaintext and Ciphertext

Given integer $m > 1$, the plaintext of BGV scheme is a cyclotomic ring $A := \mathbb{Z}[X](\Phi_m(X))$ where $\Phi_m(X)$ is the m^{th} cyclotomic polynomial. In `HElib`, plaintexts are viewed as elements to the ring A_p, where p is the plaintext prime modulus and p does not divide m. Abstracting out all the theory, `HElib` represents plaintext as **a vector of integers** modulo p, with size $d = \phi_m/(\text{ord}_m(p))$ (ϕ is the Euler's totient function, $\text{ord}_m(p)$ is the multiplicative order of p modulo m). Similarly, Ciphertext for BGV cryptosystem are also vectors of elements in the ring A.

For simplicity of showing holomorphic arithmetic, We will represent a `HElib` plaintext as integers enclosed by braces { and }. We will be representing a `HElib` ciphertext by prepending the keyword *Enc* to represent the "The encryption of".

For example, plaintext of size 48, $p = 53$, with integers 1 to 48 will be represented as follows:

$$\{1, 2, \cdots, 48\}_{53}$$

The corresponding encrypted ciphertext will be represented as

$$Enc(\{1, 2, \cdots, 48\}_{53})$$

3.2 Supported Homomorphic Arithmetic in `HElib`

Various homomorphic arithmetic is supported in `HElib`. Here We list the ones relevant to V-Oat. We are assuming a crypto-context with plaintext modulus $p = 7$, so any integer x will be changed to $x \ mod \ 7$.

1. **Addition and Subtraction**. Homomorphic addition and subtraction are supported. Parallel addition and subtraction of ciphertext and plaintext are also supported.

$$\begin{aligned} Enc(\{1,2,3,4,5\}_7) + Enc(\{1,2,3,4,5\}_7) &= Enc(\{2,4,6,1,3\}_7) \\ Enc(\{1,1,1,1,1\}_7) - Enc(\{1,2,3,4,5\}_7) &= Enc(\{0,6,5,4,3\}_7) \\ Enc(\{1,1,1,1,1\}_7) + \{1,2,3,4,5\}_7 &= Enc(\{2,4,6,1,3\}_7) \end{aligned}$$

2. **Multiplication and Power**. Homomorphic multiplication and power are supported.

$$\begin{aligned} Enc(\{1,2,3,4,5\}_7) * Enc(\{2,2,2,2,2\}_7) &= Enc(\{2,4,6,1,3\}_7) \\ Enc(\{1,2,3,4,5\}_7)^2 &= Enc(\{1,4,2,2,4\}_7) \end{aligned}$$

3. **Modulo Multiplicative Inverse.** Since p is a prime, according to Fermat's Little Theorem, for any integer a, $a^p - a$ is an integer multiple of p. This can be extended that $a^{p-1} - 1$ is an integer multiple of p, thus a^{p-1} is the modular multiplicative inverse of p. Using this fact, homomorphically evaluating the ciphertext to the power of $p - 1$ will result in encryption of 1s.

$$Enc(\{1,2,3,4,5\}_7) * Enc(\{1,2,3,4,5\}_7)^{(7-2)} = Enc(\{1,1,1,1,1\}_7)$$

4. **Total Sum**. The total sum of a ciphertext replaces all vector slots of the plaintext representation as the total sum of all integers in the vector.

$$\text{TotalSum}(Enc(\{1,2,0,0,1\}_7)) = Enc(\{4,4,4,4,4\}_7)$$

3.3 Ballot and Vote

The ballot in V-Oat system is a single ciphertext, with each position of the vector representing the corresponding candidate. For example, a ballot with 5 candidates can be represented as $Enc(\{6, 3, 1, 23, 8, 0, 0, 0, 0\}_{131})$, meaning the first candidate received 6 votes, the second received 3, and so on. The remaining zeros are padded to fit the ciphertext context because the number of candidates does not necessarily equal the size of vector plaintext.

A single vote in V-Oat system is ciphertext of 1-hot encoded vector plaintext. An example of this would be $Enc(\{0, 0, 1, 0, 0, 0, 0, 0, 0\}_{131})$. This would be that the voter voted for the 3^{rd} candidate. Note the vote and ballot need to be encrypted from the same crypto-context.

When such vote is casted, we will simply be performing a homomorphic addition of ballot and vote.

$$Enc(\{6, 3, 1, 23, 8, 0, 0, 0, 0\}_{131}) + Enc(\{0, 0, 1, 0, 0, 0, 0, 0, 0\}_{131}) = Enc(\{6, 3, 2, 23, 8, 0, 0, 0, 0\}_{131})$$

3.4 Capacity of Ciphertext

`HElib` defines the notion of ciphertext noise with the property '`Capacity`' of the ciphertext. A ciphertext has a given starting `Capacity`, and any arithmetic operation decreases the value. If this value goes below 0, the noise of the ciphertext is too big and becomes un-decryptable. `HElib` provides the function `Recrypt` as the bootstrapping method of the BGV ciphertext, which decreases the noise of the ciphertext, making it able to withstand more homomorphic arithmetic. [3]. In V-Oat, we leverage the value `Capacity` as one of vote verification methods.

4 Two-Client System Design

A major theme of V-Oat is that no participating party learns any information about vote and voter throughout the entire process. To achieve this, we designed a two-client system to spread out the responsibilities and capabilities needed in the voting procedure.

We define two types of clients of the V-Oat server: the election maintainer and the users/voters. Just like in offline voting, the election maintainer is the party that organizes the election and has the necessary information for a proper election such as the registered voters and candidates. The voters are the ones who cast their votes. V-Oat acts as the ballot and provides the 'curtains' around the ballot so that no external party could learn what goes on behind the curtains.

Figure 1 shows a visual flowchart of the voting procedure that is also outlined below:

1. **Server Initialization:** The following steps initialize V-Oat server ready for incoming voter connections.
   (a) Election Maintainer and V-Oat Server establishes TLS Connection.
   (b) The Election Maintainer provides V-Oat server the number of candidates and the number of registered Voters. The V-Oat server responds back with appropriate crypto-context parameters to support the size of the election. The Election Maintainer generates the crypto-context, public key, and secret key with the provided parameters.
   (c) Using the generated public key, Election Maintainer encrypts user information and formulates an encrypted user database
   (d) The Election Maintainer sends back the Context, Public Key, candidate information, and the encrypted User Database to V-Oat server.
   (e) V-Oat server initializes the Ballot using the provided crypto context.
2. **Voting Process:** The following steps outline the voting procedure.
   (a) User Client and V-Oat establishes TLS Connection.
   (b) User Client receives crypto context and public key.
   (c) **Client Authentication:** Upon connection, client is authenticated if they are a registered voter in the system.
      i. User encrypts its user data and sends to V-Oat

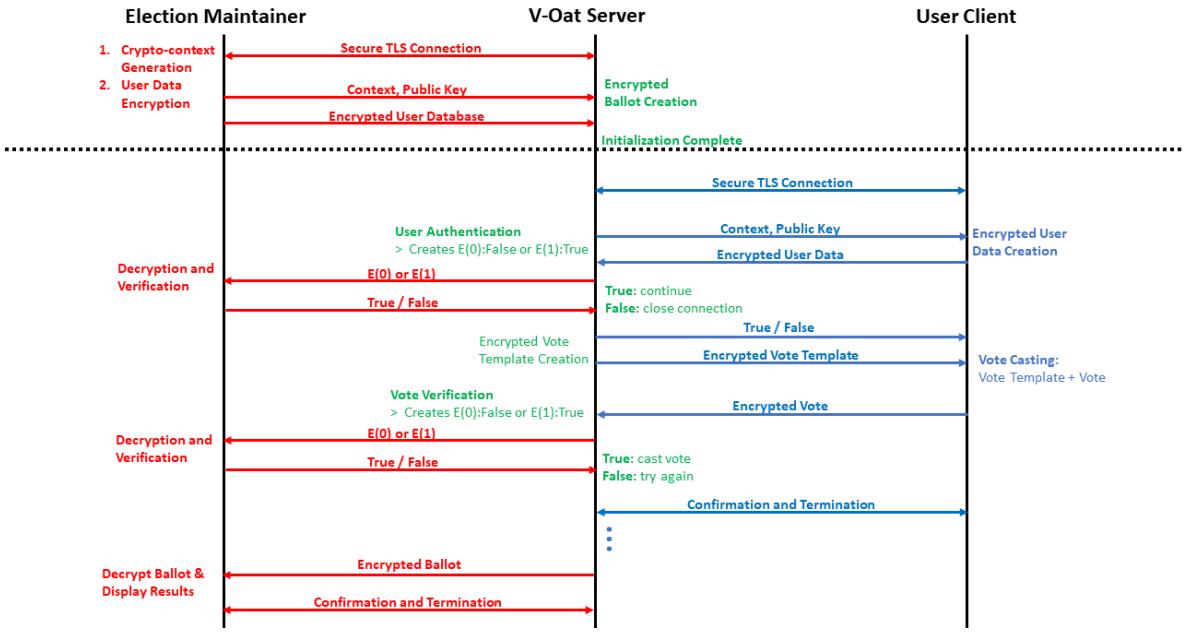

Figure 1: Visual flowchart of V-Oat

ii. V-Oat homomorphically verifies that user is a valid registered user in the database by homomorphically calculating the 'check' ciphertext. If the user is valid, 'check' ciphertext will be an encryption of 1s, or 0s otherwise.

iii. V-Oat sends 'check' ciphertext to the Election Maintainer

iv. Election Maintainer decrypts 'check' ciphertext, and returns back a boolean True/False indicating whether 'check' is indeed an encryption of 1s.

v. If V-Oat receives True, user is authenticated. If False, the user is given 5 chances to try authentication, and upon using the 5 chances, the connection closes.

(d) **Voting:**

i. Server creates the ciphertext (`Vote Template`), a fresh ciphertext with random integer values in vector slots.

ii. Server sends serialized crypto-context and (`Vote Template`) to Client over TLS sockets

iii. Client deserializes all information, adds a plaintext voting vector to (`Vote Template`), and sends back the updated version (`Received Vote Template`)

(e) **Vote Verification:**

i. Upon receiving the encrypted vote, V-Oat proceeds to verify the vote is 'valid'. It homomorphically evaluates another 'check' ciphertext, which is an encryption of 1s if vote is 'valid' and 0s otherwise

ii. V-Oat sends the 'check' ciphertext to Election Maintainer

iii. Election Maintainer decrypts and determines the validity of 'check'. Sends back True/False appropriately

iv. If True, proceed to casting. If False, the User Client is prompted again to send another vote in.

(f) The (`Encrypted Vote`) = (`Vote Template`) - (`Received Vote Template`). Cast vote by adding the actual vote to ballot: `Encrypted Ballot` + `Encrypted Vote`

(g) Upon successful casting, user client and V-Oat server connection is closed.

3. **Displaying Election Results**

(a) Upon reaching the election deadline, the ballot is closed. V-Oat server sends the ballot over to the Election Maintainer.

(b) Election Maintainer decrypts Ballot and displays results

(c) Connection closes.

5 User Verification

5.1 Encrypted User Database

We define a simple user database with two fields: username and password. Each user would have a unique username paired with a corresponding password. The username and password are strings with ASCII characters, any character values in the range 48 to 122 (inclusive) are allowed. There is a length limit of 20 characters.

The username and password strings are ACSII encoded character-by-character, and inputted in a single plaintext slot. For example, the string "user_One" will have the following plaintext representation

$$\{117, 115, 101, 114, 95, 79, 110, 101, 0, 0, ...\}_p$$

Note that plaintext modulus p needs to be greater than 122 in order to support ASCII characters up to 122. We will simply encrypt these plaintexts and pair them correspondingly.

Therefore, the encrypted user database will be a vector of pairs of ciphertexts, where the first of the pair represents encrypted username string and the second of the pair represents encrypted password string. The encrypted user database for n users is represented as the following, where $E(u_i)$ and $E(p_i)$ is the encrypted username and password for ith user.

$$[(E(u_1), E(p_1)), (E(u_2), E(p_2)), (E(u_3), E(p_3)), ..., (E(u_n), E(p_n))]$$

The user encodes and encrypts the username and password upon each authentication attempt and sends them in.

5.2 Verification Procedure

We want to make sure that the user (1) provides matching username and password pair in the database and (2) only votes once. To ensure property (1), the following procedure is done every verification attempt where the user sends in `Encrypted Username` and `Encrypted Password`.

```
EncUserDB, RecEncUsername, RecEncPassword; //database and received information
Vector<Ciphertext> checker;

foreach (EncUsername in Pair in EncUserDB):
    matchUsername = EncUsername
    matchUsername -= RecEncUsername;
    matchUsername^(p-1);
    TotalSums(matchUsername);
    matchUsername^(p-1);       // E(0) if match, E(1) if no match
    Negate(matchUsername);     // E(0) if match, E(-1) if no match
    matchUsername + 1;         // E(1) if match, E(0) if no match
    checker.push_back(matchUsername);

Ciphertext checkUsername = SUM(checker); //sum of elements of vector
sendTo(Maintainer, checkUsername);
if (receiveFrom(Maintainer, boolean) is False)
    Not Verified

Ciphertext checkPassword;
for (i in 0...EncUserDB.size()):
    checkPassword += checker[i] * EncUserDB[i].password;

checkPassword -= RecEncPassword;   // E(0) if match, E(1) if no match
```

```
Negate(checkPassword);              // E(0) if match, E(-1) if no match
checkPassword + 1;                  // E(1) if match, E(0) if no match
sendTo(Maintainer, checkPassword);
if (receiveFrom(Maintainer, boolean) is False)
    Not Verified

// User is Verified
```

We first check if the received username matches any in the database. We go through a routine for each encrypted username in the database which will produce encryption of 1s if there is a match or encryption of 0s if not. We store these ciphertexts in a vector of ciphertexts called `checker`. We sum up each ciphertext stored in `checker`, which will be encryption of 1s if there was a match. We send this sum to the Election Maintainer and receive back True/False depending on whether there was a match or not.

If there was a match in username, we then create another ciphertext `checkPassword` and we sum the product of corresponding ciphertext in `checker` vector and the encrypted password in the database. Only one of the ciphertexts in `checker` should be an encryption of 1s, so the sum `checkPassword` will be a sum of encryption of 0s or encryption of ($1 * \texttt{password} = \texttt{password}$). This will result in the corresponding password to be extracted out and saved into `checkPassword`. We finally subtract (`checkPassword - Received Enc Password`), and create an encryption of 1s if they match, or 0s otherwise. We send this to Election Maintainer to receive the result.

This procedure is done fully homomorphically – at no point are we decrypting the information. Therefore, V-Oat never knows which username and password pair in the database made a match with the received one. Also, we are only sending to the Election Maintainer the products of comparison, which is encryption of 1s or 0s according to whether there was a match or not. This does not reveal any information about the voter's username and password, except for the fact that they are verified or not. Therefore, the Election Maintainer also gains no information.

Now we want to make sure that property (2) is also accounted for: voters cannot vote twice. We first naively prevent this by enforcing sequential operation in user verification part in the multi-threaded server. Voting can be done in parallel, but no two users will be authenticated in parallel. This introduces some delay in our system but ensures that no two clients will be modifying our database at the same time.

We also leverage the vector `checker` that is created after username verification. If there is a match in the username, then `checker` will have encryption of 1s on the index of the match but 0s elsewhere. For example, if there was a match for user index 3, then `checker` will look like this:

$$[E(0), E(0), E(1), E(0), ..., E(0)]$$

The catch here is that since all information is encrypted, even though encryption of 1 is in the right place, V-Oat has no way of knowing at which index. Continuing from this fact, upon user authentication, V-Oat proceeds and multiplies in parallel the `Encrypted User Database` with `checker`. This will create two ciphertext vectors, with encryption of matched username and password in the right index, and encryption of 0s elsewhere. We parallelly subtract these two vectors from `Encrypted User Database`, and our `Encrypted User Database` will be modified so that the corresponding username and password data will be 'zero-ed out'.

$$[(E(u_1), E(p_1)), (E(u_2), E(p_2)), (E(0), E(0)), ..., (E(u_n), E(p_n))]$$

Now if the same user tries to authenticate again, they will be blocked, since no such information exists in our database.

Finally, we need to prevent any attempts to breach the system by providing encryptions of 0s as username and password, so we will place those individual checks as well. We need to do this without sending the actual username and password to the Election Maintainer, so we leverage Fermat's Little Theorem. To summarize, our verification algorithm will look like this:

```
EncUserDB, RecEncUsername, RecEncPassword; //database and received information

// Verify Non-zero
```

```
RecEncUsername^(p-1);
totalSums(RecEncUsername);
RecEncUsername^(p-1);      //E(0) if zero, E(1) if not and valid
sendTo(Maintainer, RecEncUsername)
if (receiveFrom(Maintainer, boolean) is False)
    Nor Verified
// Verify Non-zero for password as well

/*
 * Other User Auth Routine...
 */

// Delete Authenticated User
for (i in 0...EncUserDB.size()):
    duplicateUsername = EncUserDB[i].username * checker[i];
    EncUserDB[i].username - duplicateUsername;
    duplicatePassword = EncUserDB[i].password * checker[i];
    EncUserDB[i].password - duplicatePassword;
```

6 Vote Verification

6.1 Valid Vote

In this section, We define what a 'valid' vote is. To do this, We first define the notion of the 'valid voting region' of the ciphertext. Given n candidates, the valid voting region is the first n slots of the plaintext vector.

Given 5 candidates and a crypto-context that gives 10 vector slots, the valid voting region would be the first 5 slots of the plaintext vecor. This is represented as the red region in a sample plaintext below.

$$\{0,0,0,0,0,0,0,0,0,0\}_p$$

Any integers outside the region is not valid, and must be ignored. Therefore, a valid vote ciphertext is one that is "a vector that is 1-hot encoded in the valid voting region". We give examples of invalid votes below, and any mixtures of them are also considered invalid:

1. Not in the valid voting region: $\{0,0,0,0,0,1,0,0,0,0\}_p$
2. Not 1-hot: $\{1,0,1,0,0,0,0,0,0,0\}_p$
3. Multiple voting: $\{2,0,0,0,0,0,0,0,0,0\}_p$

Just for clarification sake, here are the **only** valid votes:

$$\{1,0,0,0,0,0,0,0,0,0\}_p$$
$$\{0,1,0,0,0,0,0,0,0,0\}_p$$
$$\{0,0,1,0,0,0,0,0,0,0\}_p$$
$$\{0,0,0,1,0,0,0,0,0,0\}_p$$
$$\{0,0,0,0,1,0,0,0,0,0\}_p$$

Note we could also arbitrarily define the valid voting region as n random slots in the vector. In fact, choosing the first n slots all the time may lead to security issues.

6.2 Verification Procedure

We want to make sure all our votes from the client are (1) Actually from the client and (2) Valid votes. We assume point (1) is verified through the secure TLS connection we are making between server and client. What we are more concerned is whether the (`Vote Template`) sent to the client and received back has evidence of it (A) being modified only once, and (B) returning a valid vote.

We leverage the ciphertext's `capacity` to verify point (A). Upon creation, the (`Vote Template`) has a certain `capacity` level c that denotes the amount of operations it can withhold before becoming un-decrpytable. Suppose 1 arithmetic operation subtracts the amount k from the original `capacity`. We expect the client to only perform one arithmetic to the (`Vote Template`), so the received updated (`Vote Template`) from the client should have `capacity` $= c-k$. If `capacity` $< c-k$, then that means there has been multiple modifications to the ciphertext, thus is `invalid`.

To verify point (B), we go through the following steps:

```
VoteTemplate, ReceivedVoteTemplate;
VoteChecker = ReceivedVoteTemplate - VoteTemplate;

OneHotChecker = VoteChecker^(p-1);
TotalSum(OneHotChecker);
sendTo(Maintainer, OneHotChecker)
if (receiveFrom(Maintainer, boolean) is False)
    invalid;

MultipleVotesChecker = VoteChecker;
TotalSum(MultipleVotesChecker);
sendTo(Maintainer, MultipleVotesChecker)
if (receiveFrom(Maintainer, boolean) is False)
    invalid;

ValidRegionChecker = VoteChecker * {1,...,1,0,...,0};
TotalSum(ValidRegionChecker);
sendTo(Maintainer, ValidRegionChecker)
if (receiveFrom(Maintainer, boolean) is False)
    invalid;

// Vote is Valid
Vote = VoteTemplate - ReceivedVoteTemplate; // Extract Vote
Ballot += Vote; // Cast Vote
```

After extracting the vote by subtracting the original `VoteTemplate` from `ReceivedVoteTemplate`, we go through 3 tests to check that the vote is formatted the way we want. All 3 tests are series of homomorphic arithmetic, and the decrypted result of the 3 tests show **no information** about the actual vote except for the fact that they violate our validity checks. Here are more detailed information about each checkers:

1. `OneHotChecker`: This is to ensure that only 1 slot of the vote is filled. To do this, we take the modulo multiplicative inverse of the extracted vote, so that zeros stay as zeros, and any other integers are brought down to 1s. We then take the total sum of the ciphertext, and send it to the Election Maintainer to check.

2. `MultipleVotesChecker`: This is to ensure that the one hot vector indeed contains the integer 1, not any other integer. To do this, we directly take the total sum of the ciphertext, and decrypt it. Since we already know that the ciphertext was 1 hot, if that 1 hot position was indeed a 1, then the total sum must equal a plaintext vector of 1s.

3. `ValidRegionChecker`: This is to ensure that the 1 hot occurs in the valid region of the vote. To do this, we first multiply a plaintext vector of 1s and 0s, where 1s are populated in the valid voting region and 0s else. Any values outside the voting region will become 0s because they are multiplied by 0. We then take the total sum of the ciphertext, and decrypt it. We know that the extracted vote is 1-hot and the value at that one slot is indeed a 1, so if that value resides in the valid voting region, then the decrypted value must equal a plaintext vector of 1s.

Once the extracted vote passes all tests, we know that it is finally a valid vote, and we then cast the vote by adding it to the ballot. Since all operations were done homomorphically and the resulting decryption of the checkers can only equal a vector of one integer, this checker shows no information about the vote itself, except for the fact that it is a valid vote.

7 Drawbacks and Future Work

V-Oat provides a prototype for homomorphic online voting platform that ensures zero leakage of vote and voter information. There are a few realistic limitations to V-Oat.

1. Homomorphic Encryption is known to be computationally complex and massive. A bootstrappable public key with sufficient security level could very well reach sizes up to 1GB big when serialized.
2. Trust is required among the Election Maintainer, V-Oat, and User Clients. We are operating under the assumption that V-Oat's clients trust V-Oat to not leak any information and the Election Maintainer to return back with correct True/False values for verification requests.

These points provide room for improvement in V-Oat. Some possible list to maintain these limitations would include:

1. Homomorphic Encryption Improvements:
   (a) Find a way to reduce the overhead of sending HE context and public key
   (b) Find optimal HE parameters to support particular election size
2. Expansion of distributed concept:
   (a) To strengthen the notion of trust, we can incorporate more stakeholders. Under the assumption that each stakeholder cannot trust the other, each will make sure to check the others' credibility.
   (b) Crypto context or secret key in particular could be distributed among multiple stakeholders.

References

[1] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[2] Shai Halevi and Victor Shoup. Helib design principles. *Tech. Rep.*, 2020.

[3] Shai Halevi and Victor Shoup. Bootstrapping for helib. *Journal of Cryptology*, 34(1):7, 2021.